Abstract Data Types (ADTs)

- COMP1927 Computing 2 16x1
- Sedgewick Chapter 4

ABSTRACTION

- To understand a system, it should be enough to understand what its components do without knowing how
  - Watching a television
    - We operate the tv through its interface – remote control and buttons.
    - We do not need to open the tv up and see inside to use it.
- When designing a new library, it is important to understand
  - what are the abstract properties of the data types we want provide?
  - which operations do we need to create (destroy), query, and manipulate objects of these types?
  - Do we need to or want to know how FILE * is implemented? Or just HOW to use it?

ABSTRACT DATA TYPES

- A data type is ...
  - a set of values (atomic or structured values)
  - a collection of operations on those values
- An abstract data type is ...
  - an approach to implementing data types
  - separates interface from implementation
  - builders of the ADT provide an implementation
  - Users/clients of the ADT see only the interface
    - A client can not see the implementation through the interface
      - They do not know if you used an array, a linked list etc or anything else.
      - This allows the implementation to change without breaking client code.
      - Facilitates decomposing problems into smaller parts

ADTs in C

- The interface is a contract between the client and the implementation
  - Defined in the .h file
    - typedef of the ADT
    - Function prototypes fix function names and types
- The implementation is the “inner workings” of the ADT
  - Implemented in .c file/s
    - Structs – the actual representation of the data type
    - function implementations
    - static functions
    - local typedefs

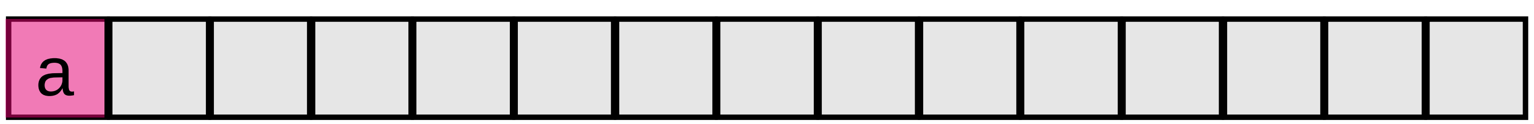
Pushdown Stack or Last-in, First-out (LIFO) Queue

- Two basic operations to manipulate a stack
  - Insert (push) a new item
  - Remove (pop) the most recently inserted item
- An operation to create a stack
  - Create an empty stack
- An operation to query the state of the stack
  - Check if stack is empty
- Applications
  - backtracking search, function call stacks, evaluation of expressions

STACK ADT IMPLEMENTATION 1: USING ARRAYS

- Array as stack
  - fill items into s[0], s[1],....
  - maintain a counter of the number of pushed items
  - pre-allocate array given maximum number of elements

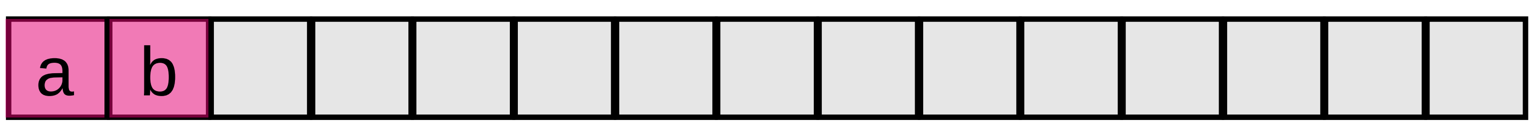
- Push a



STACK ADT IMPLEMENTATION 1: USING ARRAYS

- Array as stack
  - fill items into s[0], s[1],....
  - maintain a counter of the number of pushed items
  - pre-allocate array given maximum number of elements

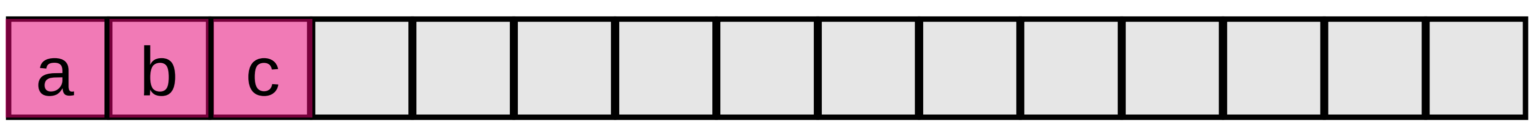
- Push a, push b



STACK ADT IMPLEMENTATION 1: USING ARRAYS

- Array as stack
  - fill items into s[0], s[1],....
  - maintain a counter of the number of pushed items
  - pre-allocate array given maximum number of elements

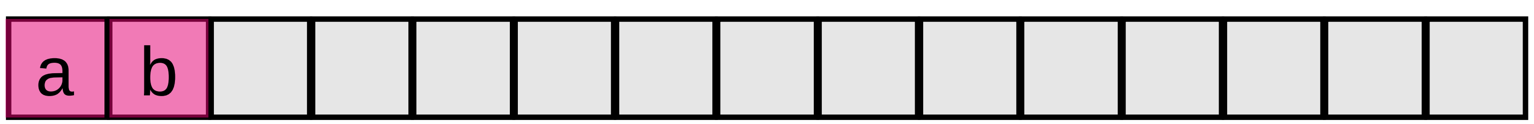
- Push a, push b, push c



Stack ADT implementation 1: using arrays

- Array as stack
  - fill items into s[0], s[1],....
  - maintain a counter of the number of pushed items
  - pre-allocate array given maximum number of elements

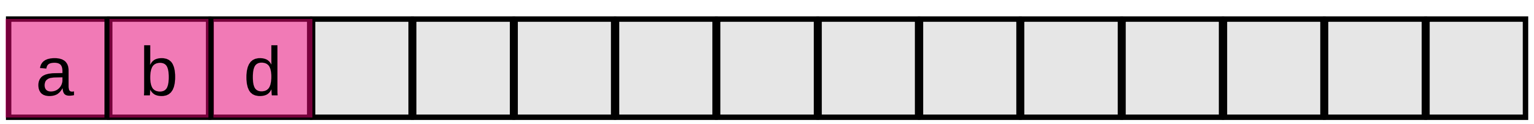
  - Push a, push b, push c, pop



STACK ADT IMPLEMENTATION 1: USING ARRAYS

- Array as stack
  - fill items into s[0], s[1],....
  - maintain a counter of the number of pushed items
  - pre-allocate array given maximum number of elements

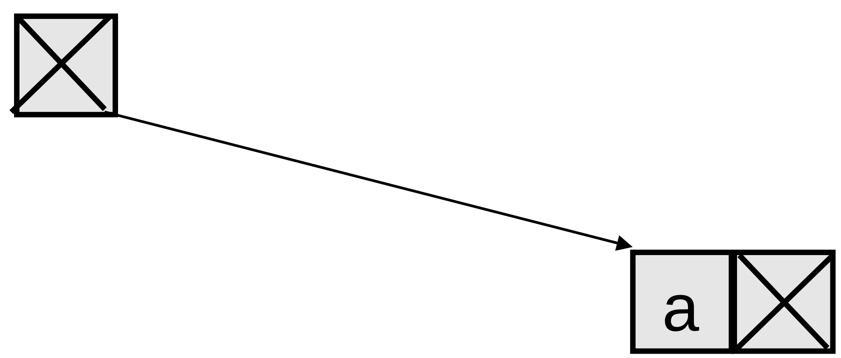
- Push a, push b, push c, pop, push d



STACK ADT IMPLEMENTATION 2: USING LISTS

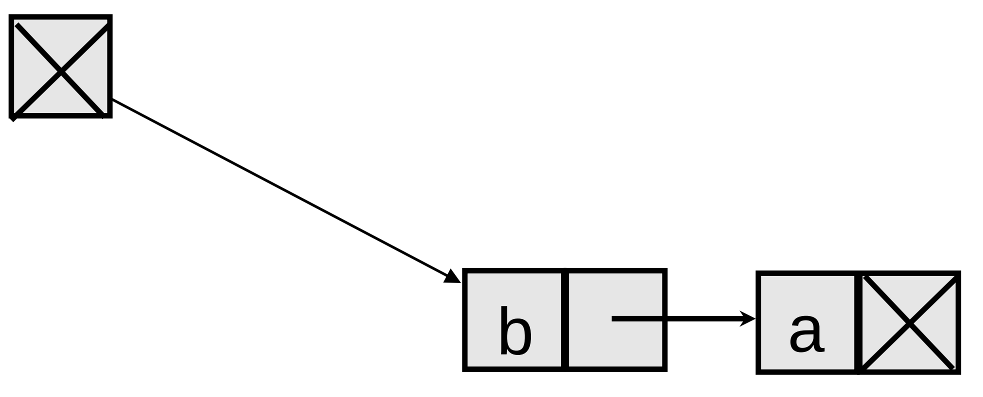
- List as stack
  - add node to front of the list when pushing
  - take node from front of the list when popping

push (a)



STACK ADT IMPLEMENTATION 2: USING LISTS

- List as stack
  - ★ add node to front of the list when pushing
  - ★ take node from front of the list when popping

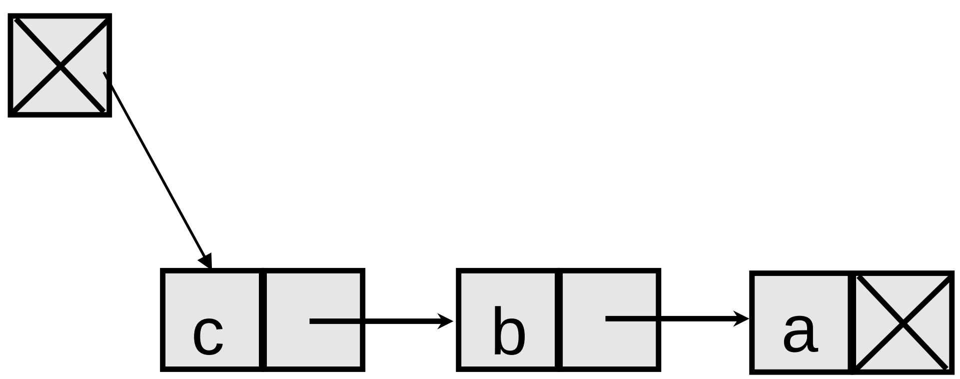


push (a)

push (b)

STACK ADT IMPLEMENTATION 2: USING LISTS

- List as stack
  - ★ add node to front of the list when pushing
  - ★ take node from front of the list when popping



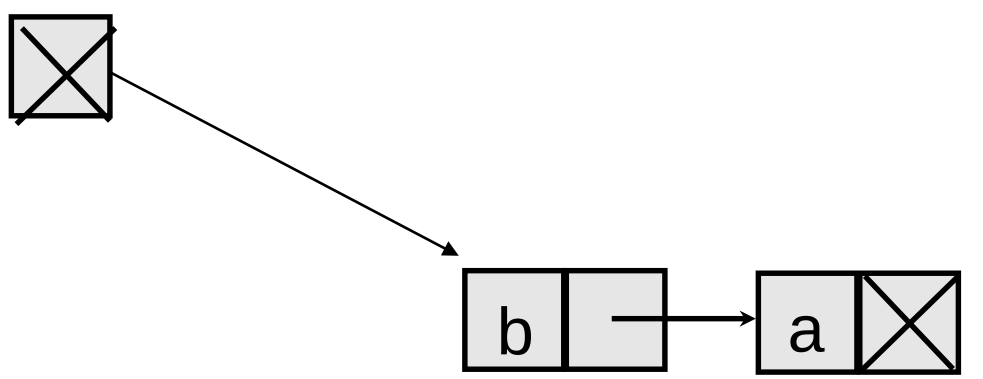
push (a)

push (b)

push (c)

STACK ADT IMPLEMENTATION 2: USING LISTS

- List as stack
  - ★ add node to front of the list when pushing
  - ★ take node from front of the list when popping



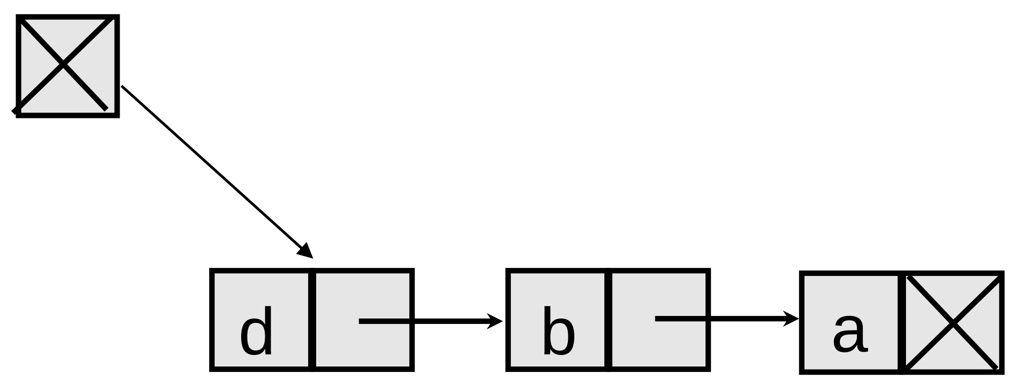
push (a)

push (b)

pop()

STACK ADT IMPLEMENTATION 2: USING LISTS

- List as stack
  - ★ add node to front of the list when pushing
  - ★ take node from front of the list when popping



push (a)

push (b)

push (c)

pop()

push (d)

Example: Balancing brackets

- Example of stack ADT use on sample input:
- ([{ }])

Next char	Stack	Check
(start)	(empty)	-
(	(	-
[	([	-
{	([{	-
}	([	{ vs }
]	(	[vs]
)	(empty)	(vs)
(eof)	(empty)	-

INFIX, PREFIX AND POSTFIX EXPRESSIONS

- Infix
- 2 + 3

- Prefix
- + 2 3

- Postfix
- 2 3 +

STACK ADT CLIENT EXERCISE: POSTFIX EXPRESSION EVALUATION

- Task: Given an expression in postfix notation, return its value:

```
% ./eval_postfix “5 9 8 + 4 6 * * 7 + *”
2075
```

How can we evaluate a postfix expression?

- We use a stack
- When we encounter a number, push it
- When we encounter an operator, pop the two topmost numbers, apply the operator to those numbers, and push the result on the stack

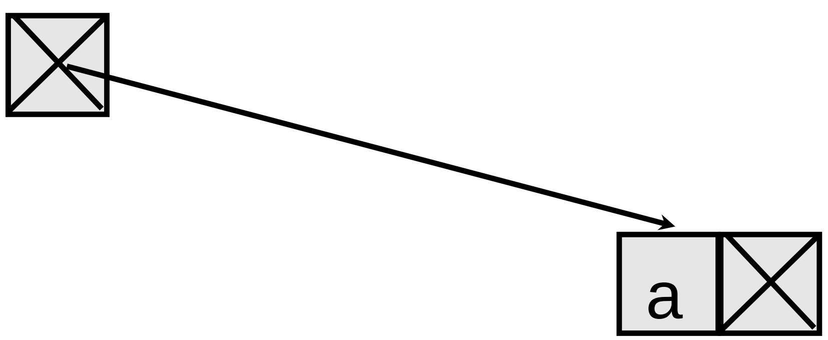
FIRST-IN, FIRST-OUT (FIFO) QUEUE

- Two basic operations to manipulate the queue
  - insert (put) new item
  - delete (get) the least recently inserted item
- An operation to create a queue
  - Create an empty queue
- An operation to query the state of the queue
  - Check if queue is empty

QUEUE ADT IMPLEMENTATION1: USING LISTS

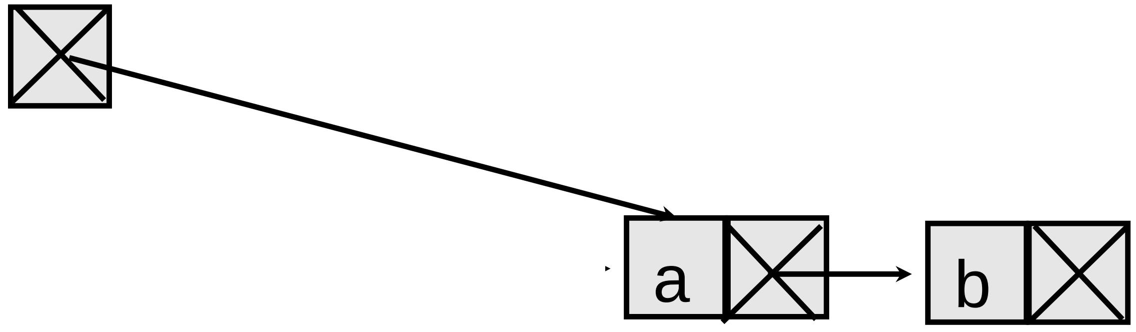
- List as queue
  - add node to end of the list when pushing
  - take node from front of the list when removing

put(a)



QUEUE ADT IMPLEMENTATION1: USING LISTS

- List as queue
  - add node to end of the list when pushing
  - take node from front of the list when removing

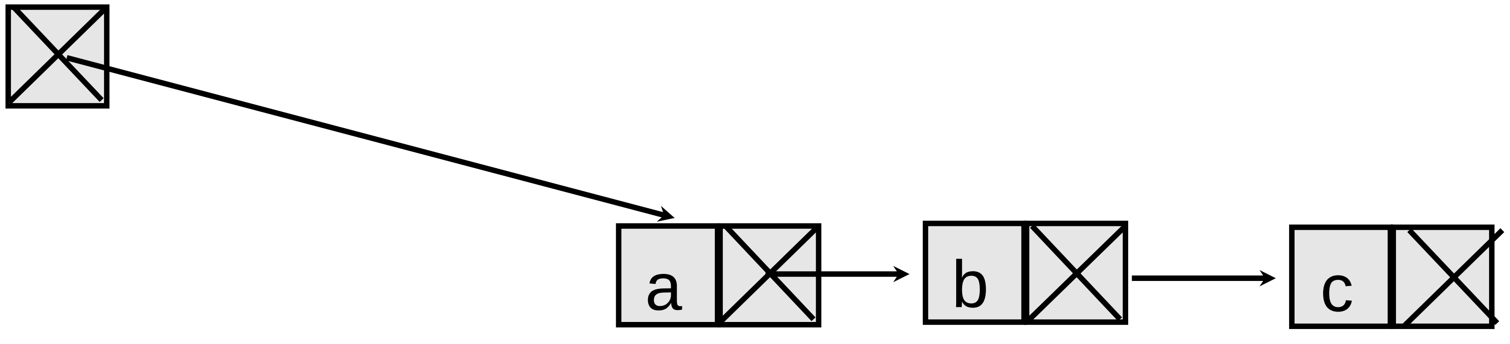


put(a)

put (b)

QUEUE ADT IMPLEMENTATION1: USING LISTS

- List as queue
  - add node to end of the list when pushing
  - take node from front of the list when removing



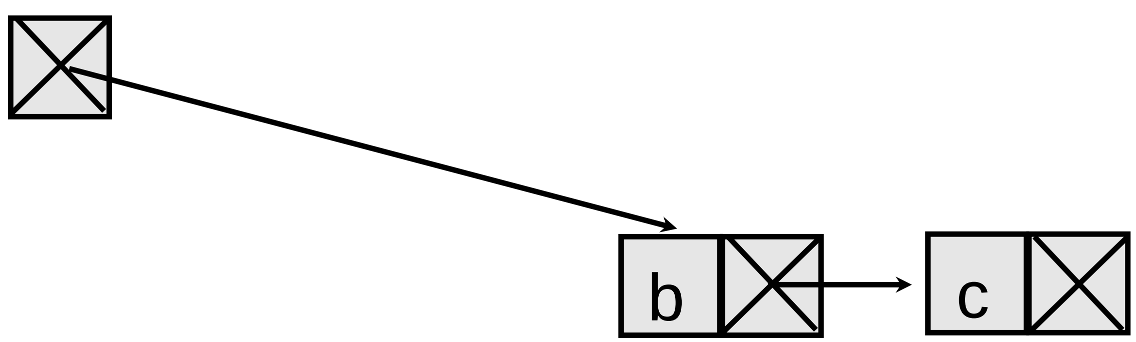
put(a)

put (b)

put (c)

QUEUE ADT IMPLEMENTATION1: USING LISTS

- List as queue
  - add node to end of the list when pushing
  - take node from front of the list when removing



put(a)

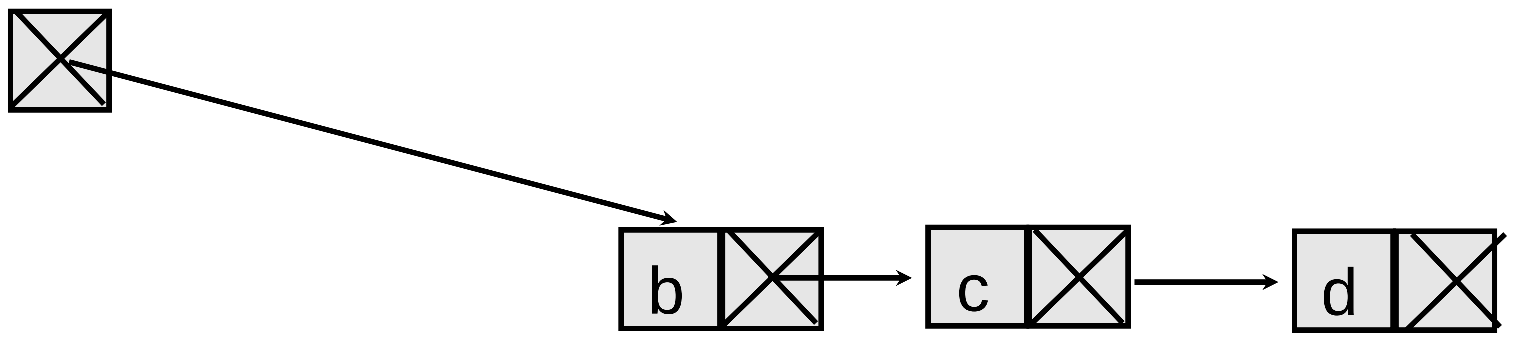
put (b)

put (c)

get()

QUEUE ADT IMPLEMENTATION1: USING LISTS

- List as queue
  - add node to end of the list when pushing
  - take node from front of the list when removing



put(a)

put (b)

put (c)

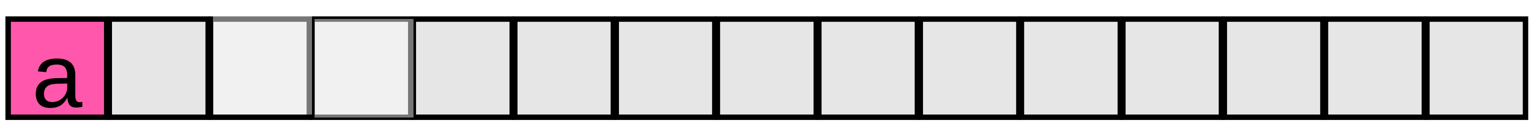
get()

put (d)

QUEUE ADT IMPLEMENTATION 2: USING ARRAYS

- Array as queue
  - fill items into s[0], s[1],....
  - maintain a counter for beginning and end of queue
  - pre-allocate array given maximum number of elements
  - roll over when reaching end of array

put(a)

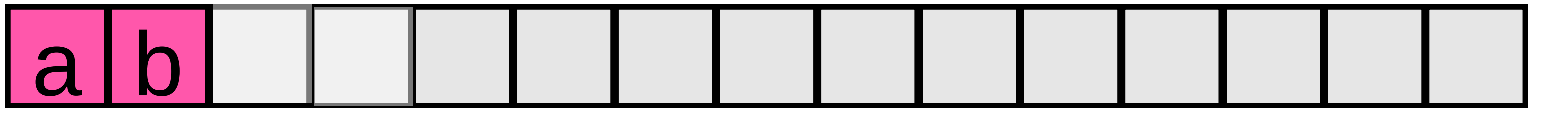


QUEUE ADT IMPLEMENTATION 2: USING ARRAYS

- Array as queue
  - fill items into s[0], s[1],....
  - maintain a counter for beginning and end of queue
  - pre-allocate array given maximum number of elements
  - roll over when reaching end of array

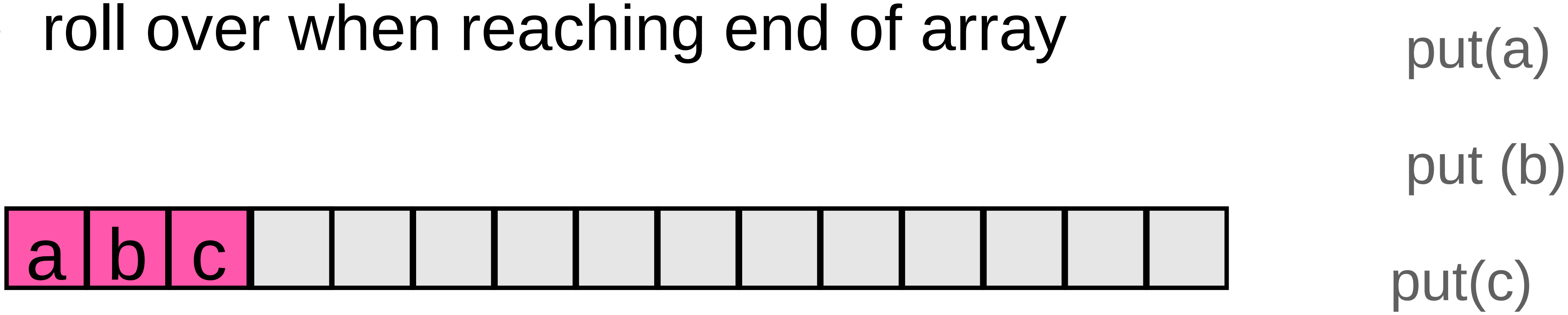
put(a)

put (b)



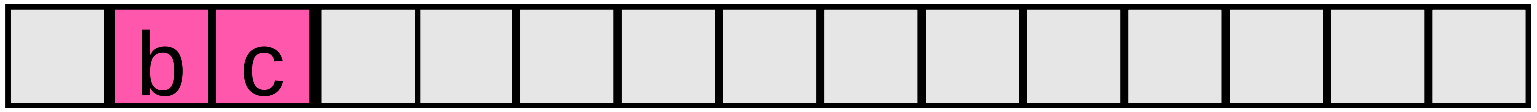
QUEUE ADT IMPLEMENTATION 2: USING ARRAYS

- Array as queue
  - fill items into s[0], s[1],....
  - maintain a counter for beginning and end of queue
  - pre-allocate array given maximum number of elements
  - roll over when reaching end of array



QUEUE ADT IMPLEMENTATION 2: USING ARRAYS

- Array as queue
  - fill items into s[0], s[1],....
  - maintain a counter for beginning and end of queue
  - pre-allocate array given maximum number of elements
  - roll over when reaching end of array



put(a)

put (b)

put(c)

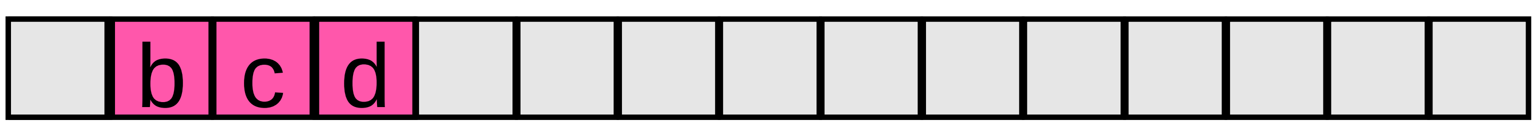
get()

QUEUE ADT IMPLEMENTATION 2: USING ARRAYS

- Array as queue
  - fill items into s[0], s[1],....
  - maintain a counter for beginning and end of queue
  - pre-allocate array given maximum number of elements
  - roll over when reaching end of array

put(a)

put (b)



put(c)

get()

put(d)

TESTING

- Testing cannot establish that a program is correct
  - would need to show for all possible inputs it produces the correct output
    - This is impossible except in trivial cases
- We can only choose this subset well!
- Different types of parameters require different types of testing
  - numeric: check the value, +ve, -ve, 0, large values, boundary cases etc.
  - string: check the length, empty, 1 element, many elements
  - properties like increasing order, decreasing order, random order

EXERCISE

- Think of some test cases for finding the maximum in an un-ordered array

Black Box vs White Box Testing

- Black Box Testing:
  - Testing code from the outside:
    - Checks behaviour
    - Does tested input result in the correct output ?
  - Program does not know about the underlying implementation
    - If the implementation changes the tests should still pass

- White Box Testing:
  - Testing code from the inside:
    - Checks code structure
    - Tests internal functions
  - Tests rely on and can access the implementation

Assert Based Testing

- How to use assert:
  - use while developing, testing and debugging a program to make sure pre- and postconditions are valid
  - not in production code!
  - it aborts the program, error message useful to the programmer, but not to the user of the application
- Use exception handlers in production code to terminate gracefully with a sensible error message (if necessary)